Lab 2: HMM's and You

ELEN E6884/COMS 86884: Speech Recognition

Due: October 17, 2005 at 12:01am

1 Overview

Hidden Markov Models are a fundamental technology underlying almost all of today's speech recognition systems. They are simple and elegant, and yet stunningly powerful. Indeed, they are often pointed to as evidence of *intelligent design* as it is deemed inconceivable that they evolved spontaneously from simpler probabilistic models such as multinomial or Poisson distributions.

The goal of this assignment is for you, the student, to implement the basic algorithms in an HMM/GMM-based speech recognition system, including algorithms for both training and decoding. For simplicity, we will use individual Gaussians to model the output distributions of HMM arcs rather than mixtures of Gaussians, and the HMM's we use will not contain "skip" arcs (*i.e.*, all arcs have output distributions). For this lab, we will be working with isolated digit utterances (as in Lab 1) as well as continuous digit strings.

The lab consists of the following parts, all of which are required:

- **Part 1: Implement the Viterbi algorithm and Gaussian likelihood evaluation —** Given a trained model, write algorithms for finding the most likely word sequence given an utterance.
- **Part 2: Implement most of the Forward-Backward algorithm —** Write the forward and backward algorithms needed for training HMM's, and test them by training the transition probabilities of an HMM.
- **Part 3: Implement Gaussian training within the Forward-Backward algorithm —** Add the updating of observation probabilities to Part 2.
- **Part 4: Train a model from scratch, and evaluate it on various digit test sets**

All of the files needed for the lab can be found in the directory `~stanchen/e6884/lab2/`. Before starting the lab, please read the file `lab2.txt`; this includes all of the questions you will have to answer while doing the lab. Questions about the lab can be posted on Courseworks (`https://courseworks.columbia.edu/`); a discussion topic will be created for each lab.

Note: The hyperlinks in this document are enclosed in square brackets; you need an online version of this document to find out where they point to.

Please make liberal use of the Courseworks discussion group for this lab, as judging from last year, it's a toughie.

2 Part 1: Implement the Viterbi algorithm and Gaussian likelihood evaluation

In this part, you will be implementing the interesting parts of a simple HMM *decoder*, *i.e.*, the program that computes the most likely word sequence given an utterance. We have pre-trained the transition and observation probabilities of an HMM on data consisting of isolated digits, and this is the model you will be decoding with.

2.1 Background

As discussed in the slides for Lecture 5 (10/6), we can use an HMM to model each word in a vocabulary. For the lab, we use the same HMM topology and parameterization as in the slides. That is, for each word, we compute how many phonemes its canonical pronunciation has, and use three times that many states plus a final state. We line up the states linearly, and each (non-final) state has an arc to itself and to the next state. We place output distributions on arcs (as in the slides) rather than states (as in the readings), and each output distribution is modeled using a GMM. For each state, the GMM on each of its outgoing arcs is taken to be the same GMM.

For example, consider the word `TWO` whose canonical pronunciation can be written as the two phonemes `T UW`. Its HMM contains $2 \times 3 = 6$ states plus a final state. Intuitively, the GMM attached to the outgoing arcs of the first state can be thought of as modeling the feature vectors associated with the first third of the phoneme `T`, the second state's GMM models the second third of a `T`, etc. The topology of the HMM can be thought of as accommodating one or more feature vectors representing the first third of a `T`, followed by one or more feature vectors representing the second third of a `T`, etc.

Now, we can use word HMM's to perform isolated word recognition in the same way that recognition is performed with DTW. Instead of a training example (or *template*) for each word, we have a word HMM. Instead of computing an ad hoc distance between each word and the test example, we compute the probability that each word HMM assigns the test example, and select the word that assigns the highest probability. We can compute the total probability an HMM assigns to a sequence of feature vectors efficiently using dynamic programming.

However, this approach doesn't scale well, and we can do better using an alternate approach. Instead of scoring each word HMM separately, we can build one big HMM consisting of each word HMM glued together in parallel, and keep track of which word each part of the big HMM belongs to. Then, we can use the Viterbi algorithm on this big HMM, and do backtracing to recover the highest scoring path. By seeing which word HMM the best path belongs to, we know what the best word is. In theory, doing dynamic programming on this big HMM takes about the same amount of time as the corresponding computation on all of the individual word HMM's, but this approach lends itself better to pruning, which can vastly accelerate computation. Furthermore, this approach can easily be extended beyond isolated word recognition.

For example, consider the case of wanting to recognize continuous digit strings rather than single digits. We can do this by taking the big HMM we would use for single digits, and simply adding an arc from the final state back to the start state. The HMM can now accept digit strings consisting of multiple digits, and we can use the Viterbi algorithm to find the best word sequence in the same way as we did for isolated digits. In this case, the best path may loop through the machine several times, producing several words of output. We use the "one big HMM" framework in this lab.

2.2 The Decoder

For this part of the lab, we supply much of a decoder for you, and you have to fill in a couple parts. Here's an outline of what the decoder does:

- Load in the big HMM graph to use for decoding.
- For each acoustic signal to decode:
  - Use the front end from Lab 1 to produce feature vectors.
  - Perform Viterbi on the big HMM with the feature vectors (*).
  - Recover the best word sequence via backtracing.

You have to implement the part with the (*); we do everything else. Now, as part of the Viterbi algorithm, you need to compute a total probability for each arc at each frame, consisting of the static arc probability multiplied by the probability of the associated GMM generating the feature vector at that frame. We have thoughtfully provided a function that computes this for you: `get_arc_log_prob()`. However, this function calls the function `LprObservR()` that evaluates the likelihood that a Gaussian assigns to a feature vector, which you must also fill in. We take care of handling the static arc probabilities and shunting around the acoustic feature vectors to where they need to go, so you don't need to deal with this.

2.3 The Big HMM Graph

We store the big HMM in a structure of type `GraphType`. We assume states are numbered starting from 1. To find out the number of states in a graph `graph`, you can call `graph.get_state_count()`. We assume that HMM's have a single "start" state, which can be found by calling `graph.get_start()`.

To find the outgoing arcs for a state, you can call `graph.get_out_arcs()`. (There is no easy way to find the incoming arcs for a state, but you don't need to do this for the lab.) Finally, you can call `graph.get_arc_log_prob()` to find the total (log) probability for an arc at a frame, *i.e.*, the product of the static arc probability (usually denoted a_{ij}) and the observation probability (usually denoted $b_{ij}(t)$ or some such). See the documentation in `Lab2_DP.C` for more details.

Arcs in the HMM are of type `ArcType`. To find the destination state of an arc `arc`, call `arc.get_dest_state`
Under the covers, each arc also has a transition probability index that lets us look up the static arc probability, an index specifying which GMM is attached to it, and possibly a word label. We attach a word label to the final arc in each word HMM, so that when we do backtracing, we can figure out which word HMM's we pass through. However, you don't need to directly access any of these other fields for the lab.

Aside: in addition to having a start state (the state that all legal paths much start in), an HMM can also have final states, states that legal paths must end in. In our implementation, a graph can have multiple final states, and each final state can have a "final probability" that is multiplied in when computing the probability of a path ending there. However, you don't have to worry about this, because we supply all of the code dealing with final states.

2.4 The Dynamic Programming Chart

When doing dynamic programming as in any of the three main HMM tasks (likelihood computation, Viterbi, Forward-Backward), you need to fill in a matrix of values, *e.g.*, forward probabilities or backtrace pointers or the such. This matrix of values is sometimes referred to as a *dynamic programming chart*, and the tuple of values you need to compute at each location in the chart is sometimes called a *cell* of the chart. Appropriately, we define a structure named `ChartCell` that can store the values you might possibly need in a cell, and allocate a matrix of cells for you in a variable named `dpChart` for you to fill in for the Viterbi computation.

For Viterbi, you should fill in the member `forwLogProbM` (this is a slight misnomer, since a forward probability is different than a Viterbi probability) and the backtrace pointer `backPtrM` for each cell in the chart.

One thing to note is that instead of storing probabilities or likelihoods directly, we will be storing log probabilities (base e). This is because if we store probabilities directly, we may cause numerical underflow. For more information, read Section 9.12 (p. 153) in Holmes!!! **We're not kidding: if you don't understand the concepts in section 9.12 before starting, you're going to be in a world of pain!!** (This can be found on the web site.) For example, we would initialize `forwLogProbM` for the start state at frame 0 to the logprob value 0, since $\ln 1 = 0$. If we want to set a value to correspond to the probability 0, we would set it to the logprob $-\infty$, or to the constant `zeroLogProbS` that we have provided which is pretty close. Hint: you may be tempted to convert log probabilities into regular probabilities to make things clearer in your mind, but resist the temptation! The reason we use log probabilities is because converting to a regular probability may result in an underflow.

2.5 What You're Supposed to Do

To prepare for the exercise, create the relevant subdirectory and copy over the needed files:

```
mkdir -p ~/e6884/lab2/
cd ~/e6884/lab2/
cp ~stanchen/e6884/lab2/Lab2_AM.C .
cp ~stanchen/e6884/lab2/Lab2_DP.C .
cp ~stanchen/e6884/lab2/.mk_chain .
```

Your job in this part is to fill in the sections between the markers `BEGIN_LAB` and `END_LAB` in two different functions: the function `ViterbiLab2G` in `Lab2_DP.C` implementing the Viterbi algorithm, and the function `LprObservR` in `Lab2_AM.C` implementing Gaussian likelihood evaluation. Read each of these files to see what input and output structures need to be accessed. In this lab, output distributions will be modeled with single Gaussians, not mixtures of Gaussians. In addition, we use diagonal-covariance Gaussians, so we need not store a full covariance matrix for each Gaussian, but only the covariances along the diagonal. Hint: remember that variances are σ^2, not σ!

2.6 Compiling and testing

Your code will be compiled into the program DCDLAB2. To compile this program with your code, type

```
smk DcdLab2
```

in the directory containing your source files (which must also contain the file `.mk_chain`).

To run this decoder on some utterances representing isolated digits, run

```
lab2p1a.sh
```

(This script can be found in `~stanchen/pub/exec/`, which should be on your path from Lab 0.) This shell script starts the executable DCDLAB2 located in the current directory with the appropriate flags. To start up DCDLAB2 in the debugger, add the flag `-debug` to the above line.

This script takes some HMM's and GMM's that we have trained for you, and runs the decoder using your Viterbi implementation and Gaussian evaluator on ten utterances, each containing a single digit. The big HMM, or *decoding graph*, used in this run consists of an (optional) silence HMM followed by each digit HMM in parallel followed by another (optional) silence HMM.

The "correct" output of this script can be found in the file `p1a.out` in `~stanchen/e6884/lab2/`. It's OK if your output doesn't match exactly, but it should be very, very close. In particular, look to see if the logprob for each utterance matches (*i.e.*, the basic Viterbi algorithm and Gaussian eval are correct) and the output word sequence matches (*i.e.*, the back pointers are correct).

To help with debugging, we have provided the file `lab2p1a.debug` which contains portions of the DP chart for the first utterance when running `lab2p1a.sh`. You should try to match these `forwLogProbM` values. Another hint: for frame 0 in the first utterance, the Gaussian associated with the outgoing arcs of state 1 should return a log prob of 20.273.

The instructions in `lab2.txt` will ask you to run the script `lab2p1b.sh`, which does the same thing as `lab2p1a.sh` except on a different test set.

3 Part 2: Implement most of the Forward-Backward algorithm

3.1 The Plan

For this part and the next part of the lab, we'll be making you implement the Forward-Backward algorithm for HMM/GMM training, but we'll have you do it in stages, to hopefully make it easier.

That is, in order to do meaningful decoding as in Part 1, we need to have trained HMM and GMM parameters. In particular, we need to select the following parameters: HMM arc probabilities, and Gaussian means and variances. To be specific, our decoding vocabulary consists of twelve words (`ONE` to `NINE`, `ZERO`, `OH`, and silence) consisting of a total of 34 phonemes. We use 3 HMM states for each phoneme (not including final states), giving us 102 states total. Each of these states has two outgoing arcs that we need to assign probabilities to, and a Gaussian that we need to estimate.

To see the transition probabilities and Gaussian parameters used in Part 1, look at the files `p1.tprobs` and `p1.oprobs` in `~stanchen/e6884/lab2/`. The file `p1.tprobs` holds the outgoing transition probabilities for each of the 102 states, the self-loop probability followed by the exit probability. The mapping from words to state indices is arbitrary, *e.g.*, the word `EIGHT` is assigned states 0 through 5, and the silence word is assigned states 99 to 101. (See p. 156 of the Holmes to see how to interpret transition probabilities.) In the file `p1.oprobs`, look for the <`gaussians`> line. After that line, alternating means and variances are listed for the Gaussian associated with each state's outgoing arcs.

Now, we can use the Forward-Backward algorithm to estimate these parameters. For example, we can initialize all parameters arbitrarily (*e.g.*, 0.5 transition probabilities, 0 means, 1 variances). Then, we can iterate over our training data, reestimating our parameters after each iteration such that the likelihood our model assigns to the training data is guaranteed to increase (or at least not decrease) over the last iteration.

What we do in each iteration is the following:

- Initialize all counts to 0.
- Loop through each utterance in the training data:

  - Use a front end to produce feature vectors.
  - Create an HMM by splicing together the word HMM's for each word in the reference transcript for the utterance.
  - Run Forward-Backward on this HMM and the acoustic feature vectors to update the counts.

- Use the counts to reestimate our parameters.

To get more details on exactly what counts to take or why this algorithm does something reasonable, you'll have to do the readings for once in your life.

Now, the Forward-Backward algorithm can be decomposed into two parts: computing the posterior probabilities of each arc a at each frame t, usually denoted something like $\gamma(a, t)$; and using these $\gamma(a, t)$ values to update the counts you need to update. In Part 2 of the lab, we'll have you do the $\gamma(a, t)$ computation, and we'll supply the code that uses the $\gamma(a, t)$ values to update transition counts and probabilities. (Updating transition probabilities is quite simple; just sum and normalize.) In Part 3 of the lab, we'll have you write the code for updating Gaussian counts, and the code to use these counts to update Gaussian means and variances.

3.2 What You're Supposed to Do

In this part, you will be implementing the forward and backward passes of the Forward-Backward algorithm to calculate the posterior probability of each transition at each frame, but not the statistics collection and reestimation parts of the FB algorithm. Your job in this part is to fill in the function `ForwardBackwardLab2G` in `Lab2_DP.C`.

For this lab, you will only need to calculate the posterior counts of *arcs* and not *states*, since both transition probabilities and observations are located on arcs in this lab. Again, this convention agrees with the slides from Lecture 4, but not the readings, so be careful. More precisely, you need to calculate the values $c_t(\text{tr}_{ij}|X)$ (which are another name for the $\gamma(a, t)$ values) (see slide 58, say, of lecture 4); the arguments `frm` and `posteriorCount` passed to `graph.add_count()` should be the values t and $c_t(\text{tr}_{ij}|X)$, respectively. **Note:** the function `graph.add_count()` expects a *regular* probability, not a *log* probability!

More specifically, you need to write code that fills in the forward (log) probability `forwLogProbM` for each cell in the dynamic programming chart. Then, we provide code to compute the total (log) probability of the utterance, and code that initializes the backward (log) probability `backLogProbM` for each state for the last frame in the chart. Then, you need to fill in code that does the rest of the backward algorithm, and that computes the posterior counts. You must call the routine `graph.add_count()` with these posterior counts, which forwards these counts to do transition count updating. Hint: read section 9.12.2 (p. 154) of the Holmes in the assigned readings. (Pedantic note: posterior counts below 0.001 are not forwarded for efficiency's sake.)

Your code will be compiled into the training program TRAINLAB2. To compile this program with your code, type

```
smk TrainLab2
```

To run this trainer on some utterances representing isolated digits, run

```
lab2p2a.sh
```

This script just collect counts for training HMM transition probabilities over ten single digit utterances and outputs them to the file `p2a.counts`. That is, for each arc tr_{ij}, `p2a.counts` will contain the sum of $c_t(\mathrm{tr}_{ij}|X)$ over each frame t of each utterance X in the training data. The "correct" output can be found in the file `p2a.counts` in `~stanchen/e6884/lab2/`. Again, it's OK if your output doesn't match exactly, but it should be very, very close. Notice that some of the counts are zero, because not all words in the vocabulary occur in this small training set.

To help with debugging, we have provided the file `lab2p2a.debug` which contains portions of the DP chart for the first utterance when running `lab2p2a.sh`.

Once you think you have this part working, run the script

```
lab2p2b.sh
```

This script reestimates transition probabilities on the data (while keeping observation probabilities unchanged), performing twenty iterations of the forward-backward algorithm and outputting the average logprob/frame (as computed in the forward algorithm) at each iteration. The training set is the same ten single digit utterances as before. If your implementation of the FB algorithm is correct, this logprob should always be (weakly) increasing and should look like it's converging. This script will output trained transition probabilities to the file `p2b.tprobs`.

Finally, decode some (isolated digit) test data with these trained transition probabilities (and the original *untrained* observation probabilities) by running the script

```
lab2p2c.sh
```

4 Part 3: Implement Gaussian training within the Forward-Backward algorithm

In this part of the lab, you will implement the statistics collection needed for reestimating a Gaussian distribution as well as the actual reestimation algorithm. This will involve filling in the functions `UpdateCountsRefR` and `ReestimateS` in `Lab2_AM.C`.

More specifically, at the beginning of each iteration through the training data, all of the Gaussian counts will be initialized to zero. Then, for each utterance, all of your calls to `graph.add_count()` during FB will now pass these posterior counts to `UpdateCountsRefR` to let you update Gaussian statistics. At the end of each iteration through the training data, the routine `ReestimateS` will be called for each Gaussian to let you reestimate these parameters from the counts you collected. As far as where to find the update equations, you can look at equations (9.50) and (9.51) on p. 152 of Holmes or equations (8.55) and (8.56) on p. 396 of HAH. Hint: remember to use the *new* mean when reestimating the variance. Hint: remember that we are using diagonal-covariance Gaussians, so you only need to reestimate covariances along the diagonal of the covariance matrix.

Your code will be compiled into the training program TRAINLAB2. To compile this program with your code, type

```
smk TrainLab2
```

To run this trainer on some utterances representing isolated digits, run

```
lab2p3a.sh
```

This script just collects counts for training HMM observation probabilities from the same mini-training set as before, reestimates Gaussian parameters, and outputs them to the file `p3a.oprobs`. The "correct" output can be found in the file `p3a.oprobs` in `~stanchen/e6884/lab2/`; only look at the counts after the line "`<gaussians>`". Again, it's OK if your output doesn't match exactly, but it should be quite close.

Once you think you have this part working, run the script

```
lab2p3b.sh
```

This script reestimates observation probabilities on the training set (while leaving transition probabilities unchanged), performing twenty iterations of the forward-backward algorithm and outputting the average logprob/frame (as computed in the forward algorithm) at each iteration. If your implementation of Gaussian reestimation is correct, this logprob should always be increasing and should look like it's converging. This script will output trained Gaussian parameters to the file `p3b.oprobs`. (Debugging hint: things should still converge if you update only means and not variances; by commenting out your variance update, you can see if your mean update looks like it's working.)

Decode the same test data as in the last part with this trained observation model (and *untrained* transition model) by running the script

```
lab2p3c.sh
```

By comparing the word-error rate found here with that found in the corresponding run in the last part, we can see the relative importance of transition and observation probabilities. (BTW, these error rates will be very poor since the training set is very small; there are some digits that it does not contain an instance of.)

For further evidence, run the script:

```
lab2p3d.sh
```

This script starts with the observation model in `p3b.oprobs` and trains both the observation and transition probabilities on the given training set for five iterations, creating the files `p3d.oprobs` and `p3d.tprobs`. It then decodes the same test set as before with these new models.

5 Part 4: Train a model from scratch, and evaluate it on various digit test sets

In this section, we run our trainer/decoder on some larger data sets and look at continuous digit data (consisting of multiple connected digits per utterance) in addition to isolated digits.

First, let us see how our HMM/GMM system compares to the DTW system we developed in Lab 1 on isolated digits. We created a test set consisting of 11 isolated digits from each of 56 test speakers, and ran DTW using a single template for each digit from a pool of 56 training speakers (using a different training speaker for each test speaker). This yielded an error rate of 18.8%.

Run the following script:

```
lab2p4a.sh
```

This first trains a model on 100 isolated digit utterances (with five iterations of FB), and then decodes the same test set as above; then, trains a model on 300 utterances and decodes; then, trains a model on 1000 utterances and decodes. See how the word-error rate varies according to training set size. The trained models are saved in various files beginning with the prefix `lab2p4a`.

Next, run the following script:

```
lab2p4b.sh
```

This takes the 300-utterance model output by `lab2p4a.sh` and decodes connected digit string data (rather than isolated digits) with this model. It also trains a model on 300 connected digit sequences and decodes the same test set.

6 What is to be handed in

Make a copy of the ASCII file `lab2.txt` from the directory `~stanchen/e6884/lab2/`. Fill in all of the fields in this file and E-mail the contents of the file to `stanchen@watson.ibm.com`. (Please paste this file into the main body of the E-mail; *i.e.*, don't include this file as an attachment.)